

PROGRAMMING EXPERT

FAQ

Q What is refactoring?

A This is when you rewrite code to improve structure and understandability. The idea of regular refactoring was made popular by the extreme programming movement.

Rewriting a program carries the risk of introducing bugs. One common refactoring is to change the name of a variable to something more meaningful. If you miss one use of the name, you'll introduce a bug. To make refactoring safe, it is usually combined with 'unit testing', which checks a program is doing what it is designed to. Tools to help with unit testing and refactoring are finding their way into development environments such as Visual Studio. Even the Express Editions of Visual Studio have a Refactor menu item, which helps with renaming variables and converting blocks of code into new methods.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Protecting your hard disk

Worried hard disk failure is just round the corner? **Mike James** explains how to check the health of your disk with the Windows Management Interface and build a self-monitoring monitor



It's an alarming fact that hard disks can fail suddenly and unexpectedly, giving you no chance to back up your data. Self-Monitoring and Reporting Technology (SMART) is a facility built into all modern disk drives that analyses their performance with a view to predicting imminent failure. While many drives fail without giving prior warning, a significant number show warning signs in their SMART data. An extensive study of disk failures concluded that "44 per cent of all failed drives had increased SMART counts in at least one of four specific counters". See www.usenix.org/events/fast07/tech/schroeder/schroeder_html/index.html for a related study.

Checking your disk's SMART data is a sensible measure. Unfortunately, Windows doesn't provide easy access to the raw SMART data. This makes it difficult to monitor how a drive is ageing. This month, we'll build our own SMART monitor, and along the way we'll meet the useful Windows Management Interface (WMI).

UNDER NEW MANAGEMENT

WMI provides an amazing range of useful facilities to help you find out how a PC is operating and perform system-level tasks such as automatically shutting down, restarting or managing the print queue. Sadly, the WMI is often ignored; it looks complicated and doesn't appear to be a 'mainstream' Windows API. The introduction of .NET has made it easier, however, because the Framework includes WMI classes.

WMI is a powerful facility available in versions of Windows since Windows 98. One of its main problems is that there are so many different acronyms associated with it, such as DMFT, WBEM and CIM. These all relate to different standards committees, industry bodies and versions of the technology. The common idea is that

every device, component or facility in a PC can be represented by a class. For example, there is a disk class that has methods and properties representing disk characteristics. Your PC will have an instance of the disk class for each of the drives it has installed. Each instance records information about one particular drive. You can find out about, and even modify, the current state of the disk by working with the properties and methods of its WMI object.

There are five categories of classes:

- 1 Computer hardware** This includes add-on components such as network adaptors as well as built-in hardware such as ports, the BIOS and disk drives.
- 2 Operating system** These classes manage everything to do with Windows configuration, management and current state.
- 3 Installed applications** These provide information about the state of all managed applications.
- 4 WMI service management** These involve the configuration and management of WMI itself.
- 5 Performance counters** These provide access to all of the performance data collected while the machine is running.

Confusingly, the WMI feels like a cross between a database and a collection of objects. The best way to think about it is as a database that returns objects in response to queries. The .NET Framework provides a set of classes that allow you to query the WMI database and work with the objects returned. If you're using any of the Express Editions of Visual Studio, you'll have to use online documentation at the Microsoft website to find out about the WMI classes. The Express Editions don't include documentation on such exotic topics.

USING WMI

The key to using WMI is the ManagementObject Searcher class, and the best way to find out how it works is via a simple example. Start a new Windows project in C# Express or Visual Studio and load a reference to System.Management using Add Reference on the Project menu. Then, at the start of your program, add:

```
using System.Management;
```

Place a button on the form and enter all the following code into its click event handler. First, we need a ManagementObjectSearcher:

```
ManagementObjectSearcher WMIsearch =  
    new ManagementObjectSearcher();
```

This will return a collection of WMI objects that match a search criterion stored as an ObjectQuery object in its Query property. So for example:

```
WMIsearch.Query= new ObjectQuery(  
    "Select * from Win32_DiskDrive");
```

If you know any SQL, the WMI Query Language will seem familiar, for WSQ is based on SQL. The * is a 'match all' condition so all instances of the Win32_DiskDrive object are returned. To retrieve the objects that match, we use the Get method:

```
ManagementObjectCollection Drives =  
    WMIsearch.Get();
```

The returned collection is always the same ManagementObjectCollection type irrespective of the kind of objects returned. Now we can step through the drives collection and display whichever information interests us:

```
foreach (ManagementObject Drive in  
    Drives)  
{  
    MessageBox.Show(Drive.  
        Properties["Name"].  
        Value.ToString());  
}
```

We have to retrieve the properties of the Drive object using a lookup in an array. This is the only easy way to deal with the fact that each WMI object is different and has a different set of properties and methods. An alternative way of dealing with this difficulty is to use the WMI Extension for Server Explorer, which is included with the full Visual Studio. This automatically generates classes for each of the supported WMI objects, complete with their own properties and methods. As not all WMI objects are supported, and the more basic method of accessing WMI is closer to the way scripts use WMI, we'll stick with that approach. However, if you are using a full copy of Visual Studio, by all means use the automatically generated wrapper classes.

Using WMI from .NET is a matter of creating an instance of ManagementObjectSearcher, performing a query to return WMI objects, and using the properties and methods of the objects returned. This is simple enough. The only difficulty in using WMI is finding out what the classes are, which properties and methods they support, and determining if they work as advertised. It is also worth knowing that you can

make a WQL query in one step using an alternative ManagementObjectSearcher constructor:

```
ManagementObjectSearcher  
WMIsearch =  
    new ManagementObjectSearcher(  
        "Select * from Win32_  
        DiskDrive");
```

WORKING WITH REMOTE WMI

WMI is implemented using DCOM, so it can work across a network. You can connect to another machine and read all its WMI information as easily as you can your local machine. Unfortunately, in today's security-conscious times, running remote procedures is more difficult than ever before; in fact, it can seem impossible. If you don't want to run WMI applications that work with machines across a network, skip this part as it's not pretty. However, if you do want to use remote WMI then read on; you won't easily find out how to make it work from other sources.

The .NET coding is the easy part. Every WMI query has a scope that defines where it should be executed. If you don't set a scope, it defaults to the local machine and the standard WMI information. If you want to run a WMI query on another machine, you first create a ManagementScope object:

```
ManagementScope ms = new  
    ManagementScope(  
        @"\\machine name\root\cimv2");
```

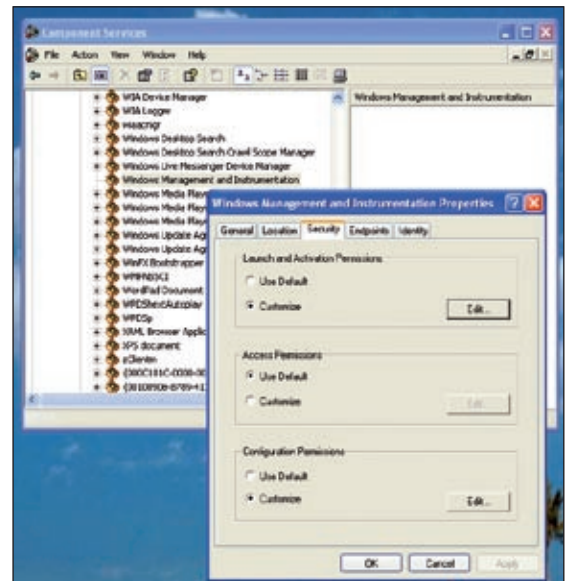
You have to replace 'machine name' with the name of the machine to which you want to connect. The 'root\cimv2' path gives the location where all the WMI objects relating to the Microsoft environment are stored. It's always a good idea to check that you can actually connect to the remote machine by first typing at the command prompt:

```
Ping "machine name"
```

You need to have the necessary security privileges to connect to a remote machine and by default you have to be an administrator, either on the local machine or a domain administrator. You can set the users and groups who are able to connect to a machine using the Component Services management tool, which you can find in Administrative tools. Expand the hierarchy until you can see DCOM Config and then find the Windows Management and Instrumentation object. Right-click, select Properties, select the Custom Tab and set Custom security for Launch, Access and Configuration to include the users or groups you want to allow to use WMI remotely. To set the credentials for the user in a program you can use:

```
ms.Options.Username = "user name";  
ms.Options.Password = "password";  
ms.Options.Impersonation =  
    ImpersonationLevel.Impersonate;
```

Here 'user name' and 'password' need to be



▲ Use Component Services to add remote WMI users

replaced by a valid identity with permission to log on to the remote machine and use WMI. The Impersonation property simply makes the remote DCOM objects that implement WMI run as if the specified user had logged on to the remote machine. It is secure in the sense that you can't do anything remotely that you couldn't do logged on to the machine directly.

SETTING AUTHENTICATION

The only remaining task is to set the authentication used – XP uses Packet and earlier systems use Connection authentication – and connect:

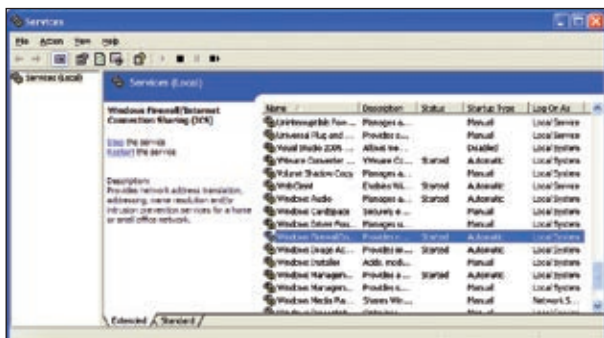
```
ms.Options.Authentication = System.  
    Management.  
        AuthenticationLevel.Packet;  
ms.Connect();
```

At this point WMI attempts to make a TCP connection with the remote machine, and nearly always fails. If it fails with an error message that says the user is unauthorised, then you need to use Component Services to edit the security settings. However, in most cases it will fail with an RPC Server not found error. The reason for this is that since Windows XP SP2 the default firewall settings block RPC, and WMI in particular.

To test if it is the firewall that is blocking remote WMI, try disabling the firewall. Run the Services Manager (which can also be found in Administrative tools), scroll down until you find Windows Firewall and then click 'stop' in the right-hand pane – you need to do this on both the remote and local machines. If you can now connect to the remote machine, the problem is the firewall; if you still can't connect, then the problem is that the machine name you are using isn't being resolved to an IP address.

There are lots of documents on the Microsoft website and elsewhere explaining how to make the firewall work with WMI/DCOM, but we have never had any success making any of these proposed methods work reliably – there are just too many security patches. This total security lock-down is becoming a real problem for application developers.

There are two ways to get WMI working. The first and simplest is to switch the firewall off – but this isn't something that many will want to



▲ Service Manager is the easiest way to turn off the firewall

consider. The second is to open the ports used by WMI. The problem is WMI uses DCOM, which, after the initial connection, randomly assigns a port number for all further communication; you can't open a random port number in advance.

The only solution we know is to use the Component Services Manager to set the range from which the random port is allocated to something small and open each one manually. To do this, start Component Services Manager again, right-click on My Computer and select Properties. Click the Default Protocols tab and double-click on the 'Connection oriented TCP/IP' entry. Use the Add button to add a suitable range of ports, say 5000-5010, and click OK.

After this, all COM+/DCOM services will select a port in this range. To open the ports, open the Windows Firewall from the Control Panel, and add one entry for each port in the range 5000 to 5010 and one for port 315, which is the port used for the initial connection. All ports are TCP. You have to open port 315 on the local and the remote machine, but the random ports need to be open only on the remote machine. After a restart, which is the only way to reinitialise the RPC service, you should find it all works.

Once you have a connected Management Scope object, you can set it as the Scope of a ManagementObjectSearcher and proceed as before, only now retrieving information from the remote machine:

```
ManagementObjectSearcher WMIsearch =  
    new ManagementObjectSearcher(  
        "SELECT * FROM Win32_  
        OperatingSystem");  
WMIsearch.Scope = ms;  
ManagementObjectCollection PCnames =  
    WMIsearch.Get();
```

GETTING SMART

Now we have the basics of WMI in .NET worked out, it's time to get back to SMART. Some information presented by WMI is provided by SMART, and this is easy to get at. For example, if you want to discover what SMART thinks of each drive, you can use the WMI Win32_DiskDrive object and its Status property. This gives an overall disk status based on SMART and other considerations. The possible values are OK, Error, Degraded, Unknown, Pred Fail, Starting, Stopping, Service, Stressed, NonRecover, No Contact and Lost Comm. Pred Fail, which is short hand for Predicted Failure, is the result we are most interested in using as a way of avoiding trouble before it happens.

Start a new project and place a button labelled 'Get Drive Status' on the form along with a RichTextBox. To read the Status property

and present it, enter the following into the button's click event handler:

```
private void button1_  
Click(object sender,  
EventArgs e)  
{  
    ManagementObject  
    Searcher WMIsearch =  
        new Management  
        ObjectSearcher(  
            "Select * from  
            Win32_DiskDrive");
```

```
ManagementObjectCollection Drives =  
    WMIsearch.Get();  
richTextBox1.Text = "Drive\t\t\  
tStatus\n";  
foreach ( ManagementObject Drive in  
    Drives )  
{  
    richTextBox1.Text = richTextBox1.  
        Text +  
        Drive.Properties["DeviceId"].  
        Value.ToString()+"\t";  
    richTextBox1.Text = richTextBox1.  
        Text +  
        Drive.Properties["Status"].  
        Value.ToString()+"\n";  
    }  
}
```

Here '\t' is the RTF code for a tab character, and '\n' is a new line.

If you want to go beyond this Predicted Failure test, there are several problems. Four WMI objects provide access to SMART, but lack of documentation makes them difficult to use. The only information available can be found at www.microsoft.com/whdc/archive/smartdrv.mspx. This describes four objects:

```
MSSStorageDriver_FailurePredictStatus  
MSSStorageDriver_FailurePredictData  
MSSStorageDriver_FailurePredictEvent  
MSSStorageDriver_FailurePredictFunction
```

The first provides information on the state of the drive similar to the state property described. The second gives the SMART data this decision was based on; the format used varies according to the make of the drive, and no information is given on how to decipher it. The third implements events based on the SMART status. The fourth provides methods that can be used to configure a SMART drive and initiate a test.

SETTING THE SCOPE

To get at the SMART data, we need to use MSSStorageDriver_FailurePredictData. This is stored in \root\wmi rather than the default \root\cimv2, so we need to set the scope:

```
WMIsearch.Scope=new  
    ManagementScope(@"\root\wmi" );
```

You can also specify a machine and user if you want to collect data about a remote machine. The WQL query is:

```
WMIsearch.Query=new ObjectQuery(  
    "Select * from MSSStorageDriver_  
    FailurePredictData");
```

```
ManagementObjectCollection  
    FailDataSet = WMIsearch.Get();
```

The FailDataSet contains a block of SMART data for each drive in the system that supports it. Each of the FailData objects in the collection has a VendorSpecific property, which returns a byte array full of SMART data.

The problem lies in figuring out what it all means, as it is really vendor-specific. The data is organised into 12-byte blocks of attribute data. The first byte of the array gives the number of attribute blocks. Each block has the format:

Item	Data
0 and 1	Unknown usually zero
2	Attribute
3	Status
4	Unknown usually zero
5	Value
6	Worst
7,8	Raw Value
9,10,11	Unknown usually zero

The attribute codes are also not standardised, but there is a core set on which you can rely:

Code	Attribute
1	Raw Read Error Rate
2	Throughput Performance
3	Spin Up Time
4	Start/Stop Count
5	Reallocated Sector Count
6	Read Channel Margin
7	Seek Error Rate
8	Seek Time Performance
9	Power On Hours Count
10	Spin Retry Count
11	Calibration Retry Count
12	Power Cycle Count
192	Power-off Retract Count
193	Load Cycle Count
194	Temperature
196	Reallocation Event Count
197	Current Pending Sector Count
198	Off-line Scan Uncorrectable Sector Count
199	Ultra DMA CRC Error Count
201	Soft Read Error Rate
220	Disk Shift

Not all drives support all attributes, and some will report attributes not in this list. To get and decode data returned by WMI, we need another RichTextBox and some appropriate headings:

```
richTextBox2.Text = "Unknw\tUnknw\  
tAttribute  
tStatus\tUnknw\tValue\tWorst\tRaw\  
t\tUnknw\n";
```

To display each set of FailData, we need a foreach loop:

```
foreach ( ManagementObject FailData  
    in FailDataSet )  
{
```

The data is returned as an object type, but we know it's a byte array. To work with it, we must retrieve it and cast it to a byte array:

```
Byte[] data = (Byte[])FailData.  
    Properties["VendorSpecific"].  
    Value;
```


Unknw	Attribute	Status	Unknw	Value	Worst	Raw	Unknw
16	0	1	15	0	200	200	0
0	0	3	3	0	230	223	203
0	0	4	50	0	100	100	52
0	0	5	51	0	200	200	0
0	0	7	15	0	200	200	0
0	0	9	50	0	95	95	243
0	0	10	19	0	100	253	0
0	0	11	18	0	100	253	0
0	0	12	50	0	100	100	52
0	0	190	34	0	64	43	36
0	0	194	34	0	111	90	36
0	0	196	50	0	200	200	0
0	0	197	18	0	200	200	0
0	0	199	18	0	200	200	0

Drive	Status
\\.\PHYSICALDRIVE0	OK
\\.\PHYSICALDRIVE2	OK
\\.\PHYSICALDRIVE4	OK
\\.\PHYSICALDRIVE1	OK
\\.\PHYSICALDRIVE3	OK

▲ The raw SMART data

Finally we can add each attribute to the RichTextBox, making use of the fact that each block of 12 bytes corresponds to an attribute:

```
for (int i = 0; i < data[0]-1; i++)
{
    for (int j = 0; j < 12; j++)
    {
        richTextBox2.Text = richTextBox2.
            Text +
            data[i*12+j] + "\t";
    }
    richTextBox2.Text = richTextBox2.
        Text + "\n";
}
```

If you now run the program, you'll see the raw SMART data displayed as a table. Your next task is to process it and build it into a useful reporting tool that will warn you if anything is going wrong.

USING A SMART STRUCT

If you want to work with the data in a more object-oriented way you probably need to create a SMART class, but a SMART struct also works well. A struct that holds values for the most important failure indicators would be:

```
public struct
    SMARTAttribute
```

```
Count;
public SMARTAttribute
    CurrentPendingSectorCount;
public SMARTAttribute
    OfflineScanUncorrectableSectorCount;
```

Packing the data into the struct is a matter of detecting which attribute the current 12 bytes of data represents. To do this, use a switch:

```
SMART smartdata;
foreach (ManagementObject FailData2
    in FailDataSet)
{
    Byte[] data2 = (Byte[])FailData2.
        Properties["VendorSpecific"].
        Value;
    for (int i = 0; i < data2[0] - 1;
        i++)
    {
        int start=i*12;
        switch(data2[start+2])
        {
            case 1:
                smartdata.RawReadErrorRate.
                    value =
                    data2[start + 5];
                smartdata.RawReadErrorRate.
                    rawvalue=
                    data2[start+7]+256*data2
```

```
[start+8];
                break;
            case 5:
                smartdata.ReallocatedSector
                    Count.value =
                    data2[start + 5];
                smartdata.ReallocatedSector
                    Count.rawvalue =
                    data2[start + 7] + 256 *
                    data2[start + 8];
                break;
            case 196:
                smartdata.ReallocationEvent
                    Count.value =
                    data2[start + 5];
                smartdata.ReallocationEvent
                    Count.rawvalue =
                    data2[start + 7] + 256 *
                    data2[start + 8];
                break;
            case 197:
                smartdata.CurrentPendingSector
                    Count.value =
                    data2[start + 5];
                smartdata.CurrentPendingSector
                    Count.rawvalue =
                    data2[start + 7] + 256 *
                    data2[start + 8];
                break;
            case 198:
                smartdata.OfflineScan
                    UncorrectableSectorCount.
                    value =
                    data2[start + 5];
                smartdata.OfflineScan
                    UncorrectableSectorCount.
                    rawvalue =
                    data2[start + 7] + 256 *
                    data2[start + 8];
                break;
        }
    }
}
```

You can add extra code to initialise the other data values and retrieve other attributes if need be. Once you see how it's done, it's not so hard. **CS**

APRESS

Extreme NXT: Extending the Lego Mindstorms NXT to the Next Level

★★★★★

£24

If you thought of Lego as plastic bricks, think again. The NXT is a microcontroller that adds a brain to Lego. You can then attach a range of sensors and motors to create robots and computer-controlled gadgets. Michael Gasperi and Philippe and Isabelle Hurbain have written an excellent book on the subject. It explains the system and all its sensors, and goes on to describe how to build new sensors. There are lots of helpful pictures of how to solder the leads to sensors, plus many useful circuit diagrams.

The book is strong on hardware and seems to be aimed at the programmer who wants to get involved in soldering and building things as well as just the software. It won't teach you how to program, but it might inspire you to burn your fingers.

SUMMARY

- ✓ Informative and clearly explained
- ✗ Emphasis on hardware not software

SUPPLIER www.amazon.co.uk
 ISBN 1590598180
 PAGES 312



APRESS

Pro Ajax and the .NET 2.0 Platform

★★★★

£34

AJAX is an important technology. It has the power to make webpages more responsive and even dissolve the distinction between desktop and web applications. Daniel Woolston's book doesn't really do the subject justice. It ranges over far too many topics and fails to concentrate on the key ideas.

While it deals with simple ideas, such as how basic JavaScript works, it ignores most of the difficulties of implementing an AJAX approach. Its coverage of the .NET approach to AJAX is slightly better, but there are too many unhelpful screen dumps and overly long listings to make this a good book. The author tends to talk around the subject matter and presents listings as if they were a substitute for a clear explanation. There are far more useful books on AJAX.

SUMMARY

- ✓ Deals adequately with simple ideas
- ✗ Fails to deal with difficult ones

SUPPLIER www.amazon.co.uk
 ISBN 1590596706
 PAGES 488

